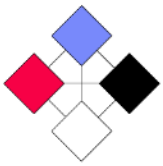
D4.1 DSL Runtime Design



DAPHNE

Integrated Data Analysis Pipelines for Large-Scale Data Management, HPC, and Machine Learning

Version 1.0

PUBLIC

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 957407.

Document Description

Integrated data analysis (IDA) pipelines, that combine data management (DM) and query processing, high-performance computing (HPC), and machine learning (ML) training and scoring, become increasingly common in practice. Interestingly, systems of these areas share many compilation and runtime techniques, and the used – increasingly heterogeneous – hardware infrastructure converges as well. Yet, the programming paradigms, cluster resource management, data formats and representations, and execution plans differ substantially.

DAPHNE is an open and extensible system infrastructure for such IDA pipelines, including language abstractions, compilation and runtime techniques, multi-level scheduling, hardware (HW) accelerators, and computational storage. DAPHNE aims for increasing productivity and eliminating unnecessary overheads.

In this report, we describe the initial design of the DAPHNE runtime system and its prototype implementation, and we discuss the support for distribution primitives and the integration with existing frameworks. This document is the result of many discussions among the consortium partners that participate in WP4 "DSL Runtime and Integration", i.e., ICCS, KNOW, UNIBAS, ETH, ITU, HPI, and UM.

D4.1 DSL Runtime Design			
WP4 – DSL Runtime and Integration			
Type of document	R	Version	1.0
Dissemination level	PU		
Lead partner	ICCS		
Author(s)	Vasileios Karakostas (ICCS), Georgios Goumas (ICCS), Stratos Psomadakis (ICCS), Konstantinos Bitsakos (ICCS), Aristotelis Vontzalidis (ICCS), Matthias Boehm (KNOW), Patrick Damme (KNOW), Mark Dokter (KNOW), Florina M. Ciorba (UNIBAS), Ahmed Hamdy Mohamed Eleliemy (UNIBAS), Gabrielle Poerwawinata (UNIBAS), Pinar Tozun (ITU), Aleš Zamuda (UM), Janez Brest (UM), Borko Bošković (UM), Milan Ojsteršek (UM), Tomaž Kosar (UM), Marjan Mernik (UM), Matej Moravec (UM)		
Reviewer(s)	Andreas Laber (IFAT), Aleš Zamuda (UM), Philippe Bonnet (ITU)		

Revision History

Version	Item	Comment	Author / Reviewer
v0.1	Initial outline	21/09/2021	Vasileios Karakostas
v0.2	Updated outline	28/09/2021	Vasileios Karakostas
v0.3	Updated content	07/11/2021	All authors
v0.4	Updated content & polished text	08/11/2021	All authors
v0.5	Updated content based on comments from Andreas Laber	15/11/2021	Vasileios Karakostas, Stratos Psomadakis, Konstantinos Bitsakos, Aristotelis Vontzalidis
v0.6	Updated content based on	16/11/2021	Vasileios Karakostas, Mark

	comments from Philippe Bonnet		Dokter
v0.7	Updated content based on comments from Aleš Zamuda and Philippe Bonnet	24/11/2021	Vasileios Karakostas, Matthias Boehm
v1.0	Final polishing	28/11/2021	Vasileios Karakostas

Table of Contents

Table of Figures

List of Tables

List of Abbreviations

Abbreviation	Meaning
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
COO	Coordinate List
CPU	Central Processing Unit
CSR	Compressed Sparse Row
CSV	Comma-seperated Values
CUDA	Compute Unified Device Architecture
DAG	Directed Acyclic Graph
DB	Database
DDP	Distributed Data-Parallel
DSL	Domain Specific Language
GPFS	General Parallel File System
GPU	General Processing Unit
FPGA	Field Programmable Gate Arrays
HPC	High Performance Computing
HDFS	Hadoop Distributed File System
IDA	Integrated Data Analysis
I/O	Input / Output
JIT	Just-in-Time
JVM	Java Virtual Machine
MKL	Math Kernel Library
ML	Machine learning
MLIR	Multi-Level Intermediate Representation
MPI	Message Passing Interface
NCCL	NVIDIA Collective Communications Library
NetCDF	Network Common Data Form

NUMA	Non-Uniform Memory Architecture
OpenMP	Open Multi-Processing
OpenBLAS	Open Source Basic Linear Algebra Subprograms
RPC	Remote Procedure Call
SIMD	Single Instruction Multiple Data
POSIX	Portable Operating System Interface

1 Introduction

Integrated data analysis (IDA) pipelines typically combine data management (DM) and query processing, high-performance computing (HPC), and machine learning (ML) training and scoring tasks. While all these tasks share similarities, their development and execution typically require the involvement of specialized frameworks, making the development of such IDA pipelines cumbersome and leaving significant room for performance improvement.

DAPHNE is an open and extensible system infrastructure for developing and executing IDA pipelines. The main objectives of DAPHNE are to: (i) increase programmer productivity, and (ii) eliminate unnecessary performance overheads. To meet these objectives, DAPHNE's approach includes language abstractions, compilation and runtime techniques, multi-level scheduling, hardware (HW) accelerators, and computational storage.

This document focuses on the DAPHNE runtime that provides the implementation of kernels for local operations, key primitives for distribution, distributed block operations on top of these primitives, integration with existing frameworks and libraries for productivity and interoperability, as well as efficient I/O and communication primitives.

1.1 Overview of DAPHNE System Architecture

Figure 1.1 shows the DAPHNE system architecture [B21, D+22]. The users specify their IDA pipelines in the *DaphneDSL*, i.e., a language similar to Julia, PyTorch, or R, or in the *DaphneLib*, i.e., a high-level Python API with lazy evaluation that internally compiles DaphneDSL scripts as well. These scripts are then compiled via a multi-level compilation chain based on the MLIR infrastructure [L+21]. The compiler converts the DaphneDSL scripts to *DaphneIR* (intermediate representation), an MLIR dialect comprising conditional control flow, matrix and frame data types, as well as logical frame and matrix operations. Then, the compiler performs multiple optimization passes and eventually lowers logical operations to kernels, i.e., physical operations that have different backend implementations for local/distributed execution and for various hardware devices. The runtime system provides implementations for the kernels, distribution primitives, and support for scheduling. The scheduling logic is responsible for deciding the size and execution order of operations and selecting the nodes and hardware devices they should be executed on. The hardware accelerators provide specialized support for offloading the execution of performance-critical operations, while the computational storage enables opportunities for combining computational and I/O kernels by shipping code execution to where data is stored, avoiding unnecessary data movement. Finally, the DAPHNE use cases demonstrate impact on real-world applications, while the DAPHNE benchmarks enable systematic performance evaluation.

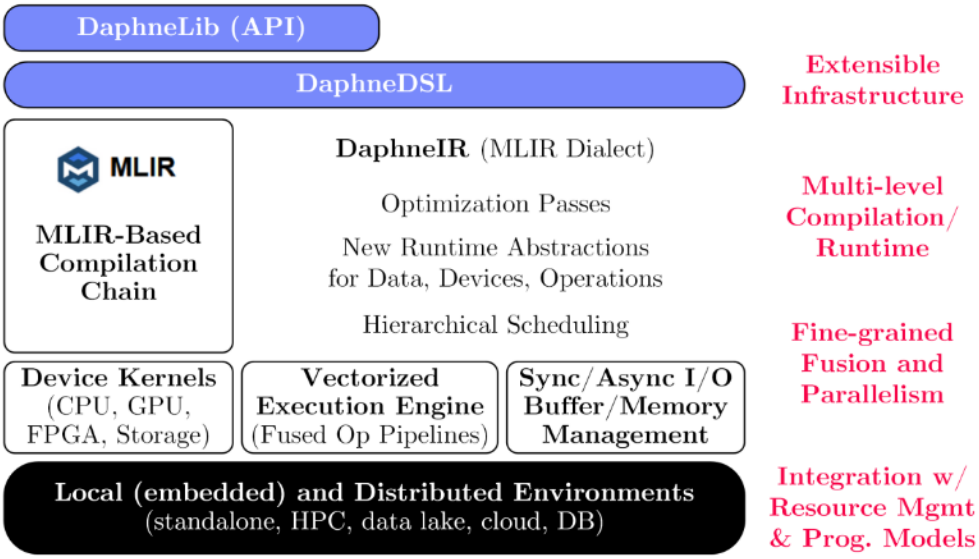


Figure 1.1.1: DAPHNE System Architecture.

1.2 Runtime Overview

The DAPHNE runtime system comprises local and distributed kernels, for implementing computational, I/O and combined operations, targeting various devices (CPUs, GPUs, accelerators, and computational storage). These kernels generally correspond to DaphneDSL operators. The MLIR-based compilation chain analyzes the user code (DaphneDSL scripts) and eventually generates calls into these kernels. The runtime also includes different implementation of the kernels for the various data structures supported by the DaphneDSL (scalars, dense/sparse matrices, and frames), and the necessary infrastructure and support for the distributed execution of kernels and the tiled[1] execution engine.

With respect to the other DAPHNE components (and corresponding work packages – WPs), the runtime system lies in the heart of the architecture, as it leverages the compiler's output (WP3: "DSL Abstractions and Compilation") to execute kernels based on the various levels of the scheduling logic (WP5: "Scheduling and Resource Sharing") on the available hardware resources (WP6: "Computational Storage" and WP7: "Hardware Accelerator Integration").

In this report, we describe the initial design of the DAPHNE runtime system and the underlying prototype implementation, and we discuss the support for distribution primitives and the integration with existing frameworks.

1.3 Terminology

To ease the understanding of this document we provide definitions for the following terms.

IR Operation is a logical operation on abstract data types (e.g., frame, matrix, scalar) which can be systematically lowered, i.e., enriched with additional properties (information about

[1] Note that we use the terms "tiled" execution engine and "vectorized" execution engine interchangeably. The definition of the term "vectorized" execution engine has been included in the deliverable D2.1 "Initial System Architecture".

output types, sizes, side effects, device placement, etc.). Ultimately, IR operations are lowered to one or multiple kernel calls.

Fused operation/pipeline combines one or more logical operations into the same operation. Fusion combined with tiled execution improves performance by allowing better spatial locality and parallelism.

Kernel is an implementation of an IR operation (or registered user-defined kernel) that operates on instantiated and materialized data types. Most kernels are stateless (except memory allocation) and deterministic. Stateful kernels are allowed as well (e.g., implementing configuration management and setup/tear-down of context objects for device/cluster initialization and cleanup). The kernels can be provided either by the DAPHNE developer that works on the development of the DAPHNE infrastructure, by external shared libraries, or by the DAPHNE user in the DaphneDSL scripts through user-defined kernels.

Fused kernels work on materialized inputs and outputs, internally setup the split and combine functions, create tasks (which in turn contain sub-programs of kernels working on tiles of the input/output), and orchestrate the workers.

Accelerator kernels are device specific kernels that have two components, the host kernel that prepares the inputs and outputs, and launches one or multiple device kernels.

1.4 Document Organization

This document is organized as follows: Section 2 describes the requirements and the challenges for the runtime system. Section 3 describes the design of the DAPHNE runtime system. Section 4 describes the basic distribution primitives that the DAPHNE runtime system will support. Section 5 discusses the integration of the DAPHNE runtime system with existing frameworks and libraries. Section 6 discusses related work, and Section 7 concludes this document and provides directions for future work.

2 Runtime Requirements & Challenges

In this section we discuss the main requirements from the perspective of the runtime system and the challenges that arise.

2.1 Requirements

The main requirements that the runtime system needs to fulfill are the following:

- ***Support for data structures***: The runtime system should support the various data structures, that the user is allowed to declare and use, and that the compiler recognizes and uses after the various compilation steps.
- ***Support for multiple operations through various kernels***: The runtime system should provide support for the execution of multiple operations that the programmer may use in DaphneDSL. These operations are translated into individual computational, I/O, and combined kernels by the compiler. The runtime system should provide the implementation of these kernels and the necessary support for executing them on the available hardware resources in case corresponding implementations exist.

- ***Support for local and distributed execution***: The runtime system should provide support for executing the operations and kernels in both local resources, i.e., using the available hardware resources within a single compute node, and distributed resources, i.e., using multiple compute nodes of a cluster. Hence, the runtime system should provide primitives and mechanisms for distributing and caching data, splitting computation tasks among multiple workers, and collecting results.
- ***Support for exploiting heterogeneous resources***: The runtime system should provide support for executing the kernels in the available heterogeneous resources that may exist within a single compute node. These resources may include multicore CPUs, typically organized in multiple sockets based on a NUMA topology, GPUs, FPGAs, and computational storage devices. The runtime system should provide support for executing the various kernels, or at least the most critical ones in terms of performance, on the available hardware resources through backend implementations of the corresponding kernels, i.e., accelerator kernels.
- ***Support for effective decision making***: The runtime system should provide support for effective and intelligent decision making that may target various optimization criteria, such as minimizing the application execution time, increasing the resource utilization of the hardware resources, or improving the system energy efficiency.

2.2 Challenges

The main challenges that need to be considered in the design and implementation of the runtime system are the following:

- ***Low overhead***: The runtime system should meet all the aforementioned requirements without affecting the application performance.
- ***Extensibility***: The runtime system should be extensible in order to easily support the execution of new operations and kernels, and the integration with other heterogeneous resources and devices.
- ***Integration***: The runtime system should handle the various forms of mismatch between the DAPHNE data structures and kernels and the underlying capabilities of the available hardware resources.

3 Runtime Design

This section describes the design of the DAPHNE runtime system and provides information regarding the current status of the prototype, where applicable. We first describe the general hierarchical approach of the DAPHNE system and then we split the description in regard to local and distributed execution.

3.1 General Hierarchical Approach

The design of the DAPHNE runtime system is based on a hierarchical approach, as shown in Figure 3.1. The user provides the DaphneDSL code to the master node of DAPHNE. The DAPHNE compiler generates the execution plan of the operations, i.e., the execution order of the corresponding kernels. In the current prototype, the compiler is responsible for deciding whether the execution of a kernel should be performed locally in the master node or distributed across the worker compute nodes, as the compiler may derive information

regarding the sizes and the characteristics of the operations' inputs and outputs and use heuristics and cost models for deciding. Later on in the project we plan to extend the DAPHNE infrastructure in order to allow for the runtime system to make such decisions on its own as well.

In case an operation is decided to be executed locally (or in case there is no cluster available), then the local runtime system orchestrates the execution on the hardware resources of the compute node.

In case an operation is decided to be executed in distributed fashion, then the runtime system of the master, i.e., the distributed runtime, is responsible for (i) deciding how to distribute the work among workers based on both the scheduling logic and the actual implementation of the distributable operations (i.e., whether they use broadcast, distribute, reduce, or other distribution primitives), (ii) sending them the code and the data, and (iii) collecting the results.

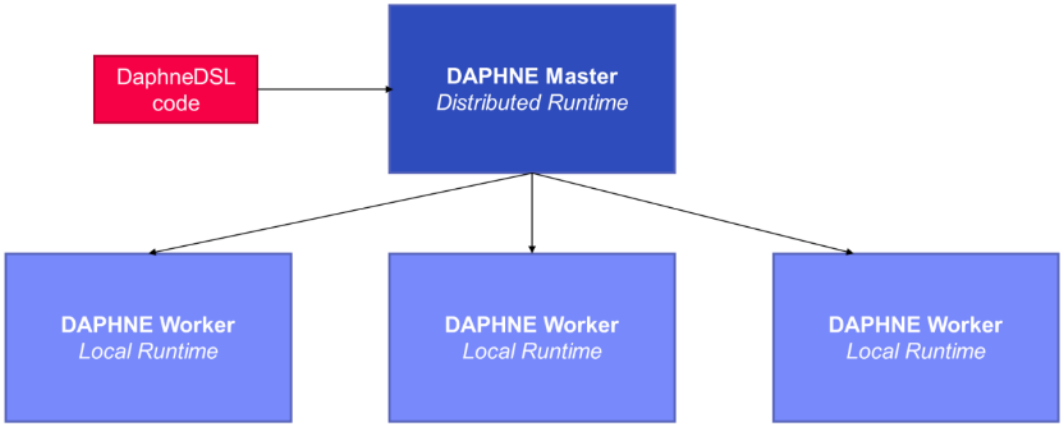


Figure 3.1: Hierarchical Design of the DAPHNE Runtime System.

3.2 Local Runtime

The local runtime is responsible for the execution of kernels in a single compute node. Note that the compute node may consist of multiple heterogeneous resources, e.g., multicore CPUs, GPUs, FPGAs, and computational storage devices.

The local runtime system consists of the following basic components:

- Data structures
- Operations and kernels
- Monitoring
- Scheduling

3.2.1 Data Structures

DAPHNE's core data structures that the programmer may use in the code beside scalars, are matrices and frames.

Matrix. DAPHNE supports both dense and sparse matrix formats. Both formats use row-major representations: for a dense matrix we use a dense linearized one-dimensional array,

and for a sparse matrix we use a compressed sparse row (CSR) [S94] format of row offsets, column-index, and value arrays.

Frame. DAPHNE also supports frames. In contrast to matrices that are homogeneous arrays, frames have a schema and thus require the handling of value types. To meet the characteristics of common analytic workloads, the DAPHNE frames rely on a column-oriented storage implemented via a dense matrix per column or column group. This composition allows the reuse of matrix operations as frame operations.

3.2.2 Operations & Kernels

DAPHNE supports the execution of multiple operations, i.e., computational and I/O operations, but also operations that combine both processing and I/O. The compiler translates these operations into kernels that are executed by the runtime. Table 1 summarizes the main categories of operations that are supported by the current prototype, along with some description and working examples.

Table 1: Information on DAPHNE operations.

Categories	Subcategories	Working examples
Data generation	Random matrices, Samples, Frames	randMatrix, fill, sample
Unary operations	Arithmetic (Absolute, Exponential), Logical, Rounding, Trigonometric operations	abs, exp, cos, sin, atan, round
Binary operations	Matrix multiplication, Arithmetic, Logical, Comparisons	@, +, *, /, -, ==
Aggregation and statistical	Full aggregation (sum, min etc.), Row-Column aggregation, Statistics for row matrices (median, quantile, etc.)	sum, min, max, mean
Data manipulation	Extract rows/columns, Insert rows/columns, Reshape/Transpose matrices, Bind matrices, Reverse/Order matrices	sliceRow, insertRow, reshape, order
Deep neural network	Activation functions, Affine, Batch Norm, Convolution, Pooling etc.	relu, softmax, conv2d, max_pool2d, batch_norm2d
Extended relational algebra	SQL queries, Set operations, Joins, Grouping	sql, merge, intersect, innerJoin, antiJoin, groupJoin
Conversions	Cast to other type, Copy	copy, quantize
I/O operations	Write to stdout, read from files etc.	print, openFile, readCsv, close

The kernels can be broadly classified into three main categories.

The first category targets computational kernels for various operations, such as data generation, unary and binary operations, aggregation and statistical operations, data manipulation, deep neural network operations and relational algebra operations among others. Note that DAPHNE supports "multiple dispatch" of IR operations and kernel calls to their "most specific" implementation, as a generalization of polymorphism to multiple objects. For example, there might be different matrix multiplication kernels for dense and

sparse matrices. The kernel dispatch can happen statically during compilation (known during compile-time), or during runtime based on the input types.

The second category targets I/O kernels and provides support for the DAPHNE user to read and store files, and for populating and storing data structures from and to storage, respectively. The current prototype supports the CSV format, but will support other file formats, such as CSV, Parquet, and Arrow as well.

The third category targets kernels that combine both processing and I/O; these are particularly important for the hardware acceleration devices, such as GPUs and FPGAs, in which the data movement plays a significant role in system performance, and even more for computational storage that bases its execution model on the notion of moving and executing code directly to the place where data lives. Such kernels can be either inferred and generated by the compiler or provided by the DAPHNE user through user-defined functions.

The kernels have backend implementations, i.e., accelerator kernels that are specialized for the different hardware devices. To enable the transparent execution of kernels on different devices, the kernels need to be implemented in a way that yields exactly the same results, modulo to round-off errors due to different parallelization strategies. However, this requirement does not imply that the same data representations (e.g., sparse vs dense outputs) need to be used. The scheduling logic (WP5) is responsible for deciding on which hardware device each kernel will be executed.

The runtime system provides support for running kernels in different hardware devices through *context objects*. Context objects are a means of keeping state at the runtime system around throughout the time of execution of a compiled program. One scenario, for example, where this is required is the use of an external library that requires the call of an initialization routine which gives back a handle object. Subsequent calls to the library would take this handle as a parameter. A concrete example where this is needed is the cuDNN library to accelerate deep neural network operations. After the call to `cudnnCreate(&handle);` the state of the cuDNN library is initialized and `cudnnConvolutionForward(handle,…);` can be executed. To keep the various needs for state tracking from spreading all over the DAPHNE framework, we use a DaphneContext object that is passed to operations at runtime. This "meta context" can hold arbitrary state information like the already mentioned cuDNN handles, which are stored in a more general CUDAContext object that holds the state of various libraries and devices that use the CUDA API. In addition to the operations categorized in Table 1, there are two "special" operations which are injected at compile time to create a DaphneContext in the beginning and destroy it in the end of execution.

3.2.3 Monitoring Support

The runtime system needs to provide support for monitoring the execution time of application tasks, along with information regarding the hardware resources and system-wide information as provided by the operating system and the platform performance counters. This monitoring information is necessary for modeling and characterizing the behavior of the application tasks, and eventually for driving the scheduling logic that decides when and how to execute the application tasks on the available hardware resources.

3.2.4 Scheduling & Tiled Execution Support

The runtime system needs to provide support for scheduling in order to optimize various goals, such as to reduce application execution time or increase the resource utilization. The goal of the DAPHNE system architecture is to exploit parallelism at multiple levels, e.g., parallelism through parfor loops, inter-operation parallelism so that independent operations may be executed in parallel or even in out-of-order fashion, intra-operation parallelism so that independent kernels that comprise an operation may be executed in parallel, and data parallelism through the tiled execution engine. To exploit the available parallelism at all levels and enable intelligent scheduling logic, the runtime system will need to provide adequate support. A key component for assisting scheduling from the runtime system perspective is the concept of *task queues*, where the various tasks that are ready for execution can be stored. The runtime system will provide such task queues, either a single queue per compute node or multiple queues with each targeting one hardware device, so that tasks can be executed on the hardware resources efficiently.

The DAPHNE system architecture introduces the tiled/vectorized execution engine for compiled operator pipelines of frames and matrices [B21, D+22]. The tiled execution engine brings many important advantages; it provides means of fusing operations (that may be computational and/or I/O operations), exploiting data parallelism without re-implementing each kernel, and integrating seamlessly with heterogeneous computing devices, computational storage, and distributed operations.

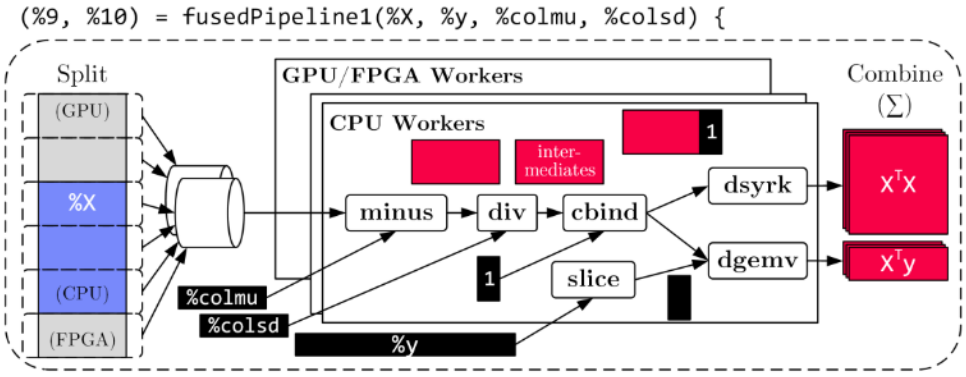


Figure 3.2: Tiled Execution of Compiled Operator Pipelines.

Figure 3.2 shows the basic integration of tiled operator pipelines into execution plans. The DAPHNE compiler is responsible for identifying the vectorizable operators, fusing them into a single vectorized pipeline[2] and generating the fused code, thus avoiding materializing intermediates. DaphneDSL defines a vectorizedPipeline operator, which the compiler then uses to call into the runtime for the actual tiled execution. The interface is the same as for any other runtime kernel. Internally, the DAPHNE runtime splits the inputs into chunks and enqueues each, along with the actual operator (this can be either a "vanilla" runtime kernel for non-fused operators or a pipeline with calls to the runtime kernels for fused operators) as

[2] The Deliverable D2.1 "Initial System Architecture" provides detailed definitions and descriptions for the terms "vectorized task" and "fused operator pipeline".

a task in a queue. It then spawns CPU worker threads which dequeue and execute the tasks from the queue. Any HW accelerator worker is implemented as a CPU thread that launches the actual accelerator kernels. Hence, the runtime system supports the tiled execution by managing the vectorized tasks, splitting the inputs and combining the outputs of the tasks, and by managing one or multiple (device-specific) task queues. The current prototype focuses on executing tasks on CPUs, and hence uses a single thread-safe FIFO queue for all CPUs. The selection of task size and execution device is part of the scheduling logic and is discussed in detail in the deliverable D5.1 "Scheduler Design for Pipelines and Tasks".

The current prototype supports parallelism through the tiled execution, as the compiler generates calls to the runtime assuming operator-level synchronization barriers. The tiled execution provides many important benefits, but it does not exploit inter-operation parallelism. To further improve performance, we plan to also support a DAG-based execution approach, where the compiler generates a DAG of the application and the runtime uses this information in order to trigger the execution of operations accordingly.

3.3 Distributed Runtime

The distributed runtime is responsible for the execution of kernels in a cluster of compute nodes. The distributed runtime system consists of the following basic components:

- Data structures
- Distribution primitives (described in Section 4)
- Distributed operations
- Communication
- Monitoring
- Scheduling

3.3.1 Data Structures

The distributed data structures that the DAPHNE compiler and runtime system use and support are the distributed collections of tiles and the federated matrices/frames. These two abstractions provide a great balance of flexibility and control.

Distributed Collections of Tiles: This representation divides a matrix into fixed-size blocks, and stores them as a collection of block-indexes and blocks [KBY17]. By default, such a bag is unordered but can be partitioned (hash, range) or sorted. Figure 3.3 shows an example 4500-x-3000 matrix, organized as a collection of squared 1K-x-1K blocks and hash-partitioned into three partitions, which can then be stored and processed in a distributed manner.

Federated Matrices/Frames: A federated matrix is a virtual matrix whose individual parts (identified by index ranges and address information for accessing remote data) are stored as local or distributed data at a federated site [B+21] or device.

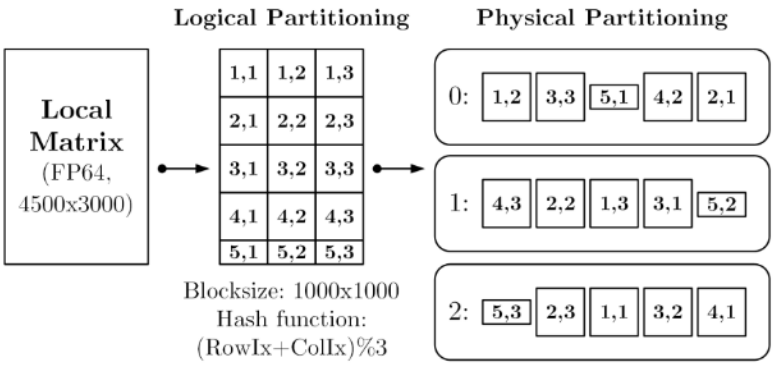


Figure 3.3: Distributed Collection of Tiles.

Figure 3.4 shows again an example 4500-x-3000 matrix that is federated across host and device memory, where the federated metadata comprises index ranges and pointers. Both of these abstractions are amenable to data-parallel computation, but they have different tradeoffs regarding distribution, load balancing, sparsity, and direct access.

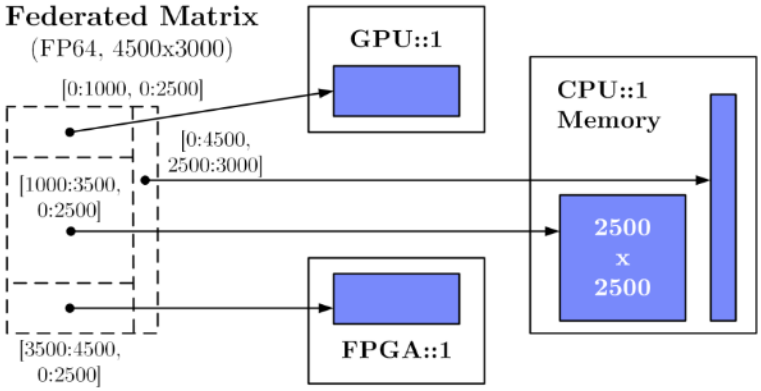


Figure 3.4: Federated Matrix.

3.3.2 Distributed Operations/Kernels

The distributed runtime system provides support for executing operations in distributed fashion, in case such implementation exists. The general flow that the distributed runtime system is based on is as follows: (i) the distributed runtime uses the distribution primitives to distribute the data and the code to the workers and waits for their completion and receiving their results, (ii) the workers receive the input data and the code, they perform the computation, and send the results back to the master node, and (iii) the master node collects the results.

In the current prototype all decisions regarding distributed execution are made at compile-time. During the first compiler pass, the compiler checks if an operation is marked as "distributable"; currently this only applies for element-wise matrix binary operations. If that is the case, each operand of the binary operation is set to be distributed by the runtime. During execution, each operand is distributed using fixed-size tiles to available workers, which are set up and started before main execution. In order to actually execute the operation, during runtime the DAPHNE master sends MLIR IR fragments (which are generated at compile-time) to each worker, that is responsible to execute the fragment on its local data. This approach allows for distributing fused pipeline operations, thus increasing data locality and reducing

data communication, and for generating and executing optimized code for each particular compute node based on its hardware capabilities. Finally, each tile is collected back by the master in order to reconstruct the resulted matrix. In the future, this can be extended to support runtime decisions regarding distribution policies and more complex collective primitives, like aggregations and reduce.

3.3.3 Communication Support

The runtime system needs to provide support for the communication between the master and the workers, and in general between the compute nodes. While various communication frameworks can be used [SCR20], we base our approach in the initial prototype for distributed compute nodes on the gRPC framework that allows both synchronous and asynchronous communication between the distributed nodes. In the future, we aim to further integrate MPI and device-specific collective operations (e.g., NVIDIA NCCL), and embedded deployments in different HPC, cloud, and database environments.

3.3.4 Monitoring Support

The distributed runtime system needs to provide support for monitoring the status of workers, i.e., DAPHNE-specific information regarding the execution of application tasks but also system-wide information as provided by the operating system and the platform performance counters, in order to leverage this information for scheduling decisions, e.g., minimize load imbalance.

3.3.5 Scheduling support

The distributed runtime system also needs to provide support for scheduling across compute nodes in order to optimize various goals, such as reduce application execution time or increase the resource utilization. Similarly to the local runtime (Section 3.2.4), the goal here is to exploit the available parallelism at multiple levels.

4 Distribution Primitives

In this section we describe the basic distribution primitives that the DAPHNE runtime system will support. In distributed systems, either for machine learning or data analysis and manipulation frameworks, a number of collective communication primitives need to be used for data exchange between executors. Note that the term "executors" may refer to distinct compute nodes but also to distinct hardware devices within a single compute node. DAPHNE uses the distribution primitives for the implementation of distributed operations and kernels. Here we present some of the most preferable communication primitives, such as Broadcast, All-Reduce, Ring-Reduce, Scatter/Gather, Prefetch, and Distributed Caching [ZC13].

4.1 Broadcast

The broadcast pattern [BCH92] is used to distribute data from one processing unit to all processing units. Broadcast can be interpreted as an inverse version of the reduce pattern. There are many implementations of the broadcast primitive.

- In generic linear broadcast, the message is split up into k packages and sent piecewise from node n to node n+1.

- In binomial tree broadcast the whole message is sent at once. After receiving the message, each node sends it on further. At each time step the amount of sending nodes is doubled. Thus, it is ideal for short messages, but falls short with longer ones, as during the time when the first transfer happens only one node is busy.
- In pipeline binary tree broadcast the algorithm combines binomial tree broadcast and linear pipeline broadcast, which makes the algorithm work well for both short and long messages. The aim is to have as many nodes work as possible while maintaining the ability to send short messages quickly. A good approach is to use Fibonacci trees for splitting up the tree, which are a good choice as a message cannot be sent to both children at the same time. This results in a binary tree structure.

4.2 All-Reduce

The concept behind All-Reduce is that data from each distributed node, is reduced in some fashion (e.g., via a sum) to produce an aggregate and this aggregate is then shared across all the nodes. Every worker shares its data with all other workers and applies a reduction operation. This operation can be any reduction operation, such as sum, multiplication, max or min. In other words, it reduces the target arrays in all workers to a single array and returns the resultant array to all workers.

There are many implementations of the All-Reduce collective primitive [P18]. Some implementations try to minimize bandwidth, while some others try to minimize latency. Next we briefly present some of them:

- Tree Reduce: The tree reduce topology [CKPT17] uses the lowest overall bandwidth for atomic messages, although it effectively maximizes latency since the delay is set by the slowest path in the tree. It is a reasonable solution for small, dense (fixed-size) messages.
- Round-Robin Reduce: In round-robin reduce [ZC13], each worker communicates with all other workers in a circular order. Round-robin reduce achieves asymptotically optimal bandwidth, and optimal latency when packets are sufficiently large to mask setup/teardown times. In practice though, this requirement is often not satisfied, and there is no way to tune the network to avoid this problem.
- Butterfly Network: In a butterfly network [K02], every node computes some function of the values from its in-neighbours (including its own) and outputs to its out-neighbours. In the binary case, the neighbours at layer-d lie on the edges of hypercube in dimension d with nodes as vertices.

4.3 Ring-Reduce

Similarly to All-Reduce, in Ring-Reduce [LLS07] data from each distributed node is reduced to produce an aggregate, which is then shared across all the nodes. In the Ring-Reduce algorithm, a ring topology of the workers/nodes is constructed. Each node divides its own data into sub-portions, which are referred to as "chunks". Each node adds its local chunk to a received chunk and sends it to the next node. In other words, every chunk travels all around the ring and accumulates a chunk in each node. After visiting all nodes once, it becomes a portion of the final result (accumulated data), and the last-visited node holds the chunk. Finally, all nodes can obtain the complete accumulated data by sharing the distributed partial

results among them. This is achieved by doing the circulating step again, but without reduction operations, i.e., merely overwriting the local chunk in each node with the corresponding received (final) chunk. The Ring-Reduce operation completes when all nodes obtain all portions of the accumulated data.

4.4 Scatter & Gather

Scatter [RD14] is a collective routine similar to Broadcast. Scatter involves a designated root node sending data to all nodes in a distributed system. The primary difference between Broadcast and Scatter is small but important. Broadcast sends the same piece of data to all nodes while Scatter sends chunks of an array to different nodes. Gather is the inverse of Scatter. Instead of spreading elements from one node to many nodes, Gather takes elements from many nodes and gathers them to one single node. This routine is highly useful to many parallel algorithms, such as parallel sorting and searching.

4.5 Prefetch

Prefetch is a general technique that may be implemented via various means, and in this case via an additional distribution primitive that allows hiding the latency of moving data across workers or devices.

4.6 Distributed Caching

Distributed cache is a distributed primitive that uses all the random-access memory (RAM) of distributed nodes into a single in-memory data store as a data cache to provide fast access to data for all nodes. Distributed cache links together multiple nodes, growing beyond the memory limits of a single node or a single physical server. Distributed cache is especially useful in environments with high data volume and load. In sophisticated architectures, distributed cache is scalable, meaning it can add nodes to its "cache cluster" in case of load increment. Modern systems and frameworks, such as Hadoop and Spark, offer the functionality of a distributed cache.

5 Integration with Existing Frameworks & Libraries

In this section we describe the integration of the DAPHNE runtime system with existing frameworks and libraries. We split the description into the front-end part and the back-end part. Note that the frameworks and libraries that are mentioned in this section are selected for broad applicability. As the development of the DAPHNE use cases (WP8) proceeds, we will reassess the list of existing frameworks and libraries for integration within DAPHNE, in order to support the use cases early on and still allow for continuous integration and development.

5.1 Frontend Integration

The frontend part refers to the existing frameworks and libraries that the DAPHNE user can use in a program written in DaphneDSL or DaphneLib through user-defined functions. Such integration provides two important benefits. First, the programmers are enabled to reuse existing code, algorithms and methods of a framework that they may be familiar with already. Second, the DAPHNE system is allowed to reuse algorithms and functionality that is already implemented in other frameworks, so that in the development of DAPHNE we avoid reimplementing preexisting components from scratch, but focus on critical components for

programmability and performance, and leverage existing frameworks for generality and applicability. To that end, we plan to support popular frameworks, such as Spark, SystemDS, TensorFlow and PyTorch, in order to reuse and leverage existing functionality related to data management and machine learning tasks.

5.2 Backend Integration

The backend part refers to the existing frameworks and libraries that the DAPHNE system internally uses during its execution. Such frameworks and libraries can be broadly categorized into the following classes:

- ***HPC libraries***, such as OpenBLAS, MKL, MPI, and OpenMP. The current prototype already uses OpenBLAS for the execution of some computational kernels; MKL can be used as an alternative drop-in replacement. OpenMP and POSIX threads (pthreads) have been integrated in the current prototype as well in order to exploit thread parallelism in multicore CPUs.
- ***Communication*** for distributed execution or across multiple devices within a compute node, such as gRPC, MPI, and NCCL. The current prototype already uses the gRPC framework for the distributed execution.
- ***Parallel file systems***, such as HDFS, GPFS, Lustre. These are currently not used (yet) by the prototype.
- ***Common data formats***, such as CSV, Parquet, Arrow, HDF5, and NetCDF. The current prototype already supports the CSV format.

6 Related Work

There are several frameworks and projects that target the domains of data-management (DM), machine learning (ML), and high-performance computing. In this section we discuss some of the most popular and relevant existing frameworks, and we describe their runtime and distributed design.

PyTorch [P+19] is an open source machine learning framework. PyTorch's runtime operates in two different ways, either in eager mode or via TorchScript [Tor21] (a highly optimized subset of Python language, which is constantly expanded). Eager mode utilizes Python's runtime to execute instructions and functions in real time as the script is being parsed. On the other hand, TorchScript compiles and generates IR code before executing at runtime. This approach enables portability, since the generated IR can be easily imported in other languages and the models can be executed in multiple devices. In addition, it allows optimizations on the IR code [PyT21a], such as: (i) Algebraic rewriting: constant folding, common subexpression elimination, dead code elimination. (ii) Out-of-Order execution: reordering operations to reduce memory pressure by making use of cache locality. (iii) Fusion: combining several operators into a single kernel to avoid overheads from round-trips to memory. (iv) Target-dependent code generation: Taking parts of the program and compiling them for specific hardware.

PyTorch also supports ease of conversion from eager mode models to TorchScript. Since this comes with some limitations, there is also the option of writing code directly in TorchScript. This option allows the serialization of the models and therefore complete independence from

the Python runtime, enabling easy deployment in servers, mobile devices, etc. There is currently a PyTorch mobile runtime under development targeted for Android and iOS devices [PyT21b].

Finally PyTorch implements a "Just-in-Time" or JIT compilation. The idea is that dynamic code might be static after being determined. Therefore the compiled version is generated at runtime, if the applied overhead is determined to be smaller than executing dynamic code. The steps of JIT are as follows: (i) specialization, e.g., determine shapes, services and other information, (ii) optimizations, e.g., algebraic rewriting, fusion, etc. and (iii) execution, e.g., scheduling, parallelism, etc.

PyTorch Distributed is a distributed backend [L+20b], which allows performance optimizations and distributed training, giving the opportunity to process more data or develop bigger models. PyTorch features two main distributed training paradigms:

Distributed Data-Parallel training (DDP) is a widely adopted SIMD training paradigm. Every process has a copy of the model, each handling a different subset of the data samples. Communication exists for updating the models during training and keeping them synced.

RPC–based distributed training is developed to support general training that cannot fit in DDP. For example, distributed pipeline parallelism (models that are too big to fit in one GPU and are broken down to different parts, which are placed on different GPUs), parameter server paradigm (where one server stores model parameters and the workers query and work on specific tasks) as well as others combinations of DDP and other paradigms.

TensorFlow Runtime (TFRT): TensorFlow [A+16] is an open source artificial intelligence library, which utilizes graphs to build models. A new runtime called TensorFlow Runtime or TFRT [TFRT21] was recently released, which is superior to the old runtime, allowing for many optimizations. Similar to PyTorch, TensorFlow allows for either eager or graph execution. On one hand, eager is executed in Python Runtime, kernels are being utilized by Python and there is no room for significant optimizations. On the other hand, graph execution code is compiled and lowered with MLIR [L+20a]. Optimizations are made and based on the available hardware, and specific kernels are used for improved performance. Early results show 28% increase in performance when compared with the old TF (1.x) runtime (which is highly optimized compared to the new one, which is still under development) [TFRT21]. In addition, the new runtime is highly extensible and modular. TFRT has a lock-free graph executor that supports concurrent operation execution with low synchronization overhead, and a thin, eager operation dispatch stack so that eager API calls will be asynchronous and more efficient. Moreover, the device runtimes are decoupled from the host runtime, the core TFRT component that drives host CPU and I/O work in order to make extending the TF stack easier. To get consistent behavior, TFRT leverages common abstractions, such as shape functions and kernels, across both eager and graph execution.

TensorFlow Distributed: TensorFlow implements an API to allow for distributed training across multiple GPUs, multiple machines or TPUs [Ten21], [A+16]. The API is designed with three goals in mind, ease of use, good performance and easy switching between different "strategies". Some types of strategies that are implemented are the following: (i)

MirroredStrategy, that targets synchronous distributed training across multiple GPUs on one machine, with the model being replicated across GPUs and updated synchronously; (ii) TPUStrategy, that targets TensorFlow training on Google's specialized ASICs/TPUs; (iii) MultiWorkerMirroredStrategy, that is very similar to MirroredStrategy but targets multiple workers, each with potentially multiple GPUs; (iv) ParameterServerStrategy that targets a training cluster consisting of parameter servers and workers, with the variables being created on parameter servers, read and updated by workers in each step; (v) CentralStorageStrategy that targets single devices or machines in which the model variables are placed on the CPU and operations are replicated across all GPUs.

Spark Runtime: Apache Spark [Z+16] is an open source, general-purpose distributed computing engine used for processing and analyzing large amounts of data. Spark uses a master/slave architecture, i.e., one central coordinator (driver) and many distributed workers (executors) [Spa21]. There are two basic ways the driver program can be run: (i) Cluster Deploy: The driver process runs as a separate JVM process inside a cluster, and the cluster manages its resources (mostly JVM heap memory). (ii) Client Deploy: The driver is running inside the client's JVM process and communicates with the executors managed by the cluster. The client process starts the driver program and then the driver and its subcomponents – the Spark context and scheduler – are responsible for requesting memory and CPU resources from cluster managers, breaking application logic into stages and tasks, sending tasks to executors, and collecting the results.

***SystemML*:** SystemML [B+16] is a flexible machine learning system that automatically scales to Spark and Hadoop clusters. In SystemML all tensors are represented as 2D matrices instead of multi-dimensional matrices in other typical deep learning applications. For example a 4-dimensional tensor [N, C, H, W] would be represented as [N, C*H*W]. This allows utilization of existing physical optimizations such as various sparse formats (COO, CSR, etc), blocking for handling out-of-core tensors and broadcasting operations over scalars and vectors. In addition, SystemML has various built-in neural network functions. To exploit the low-level CPU SIMD instructions, the SystemML runtime uses the underlying BLAS (such as OpenBLAS and Intel MKL) for compute-intensive operations such as matrix-matrix multiplication and convolution operations. If Intel MKL is installed, the SystemML runtime uses the highly-tuned MKL-DNN primitives for the convolution operations. Some other key-features of SystemML are: (i) GPU Backend: SystemML supports highly tuned kernels from CUDA libraries utilizing available GPUs. (ii) Sparse Operations: SystemML decides on runtime whether sparse or dense formats are best fit for the problem and selects appropriate operators, thus reducing the number of floating point operations and improving memory efficiency. (iii) Distributed Operations: Depending on the problem, if the user demands a lot of memory for the task, SystemML generates a distributed data-parallel plan across multiple nodes.

Finally, some MLIR projects related to machine learning and distributed runtimes are:

Nod Distributed Runtime [Nod21] provides a compiler frontend for various machine learning frameworks, like PyTorch and TF, which can efficiently parallelize and distribute

workloads on Nod's runtime allowing auto-tuned high performance execution on different hardware – from very large clusters to System-on-Chip designs.

IREE (Intermediate Representation Execution Environment) [IREE21] is a MLIR-based end-to-end compiler for machine learning models. It imports a model generated with a common ML framework and compiles and runs the model based on user's selected target platform. The IREE design is client/server, asynchronous and modular with careful consideration to use-case-driven optionality (synchronous, single-node, unity build, multi-tenant, enclave, etc.).

PlaidML [Pla21] is an advanced and portable tensor compiler. It bridges the gap between universal machine learning operations and the platform specific code needed to perform those operations with good performance. PlaidML enables deep learning on devices where the available computing power is not well supported or the available software stack contains unpalatable license restrictions.

Comet [M+21] is domain-specific programming language and compiler infrastructure for tensor contraction targeting heterogeneous accelerators. Comet specifically targets chemistry applications, which fundamentally use tensor contractions. Comet is based on MLIR, providing a multi-level IR allowing different kinds of optimizations at each level of the IR stack and therefore generating highly optimized IR code for execution.

Minos Computing Library (MCL) [G+20] is a system software for programming on extremely heterogeneous systems that increases performance and application portability. The main concept is to allow programmers to develop applications on small personal desktop machines that automatically scale up to fully utilize powerful workstations or down to power-efficient embedded systems. MCL achieves this through asynchronous execution, intelligent scheduling and efficient resource allocation. MCL consists of several components: a scheduler that orchestrates tasks, an asynchronous runtime, a well-defined API and a set of tools to analyze and debug applications. MCL serves as a backend for higher level programming models, rather than a replacement for existing high-performance computing frameworks. It provides an API through which users can create, execute, wait and check the status of a task asynchronously. The MCL runtime is a dynamic library linked to each application that implements MCL APIs and asynchronously executes tasks. Finally, regarding scheduling, in the context of MCL, which is meant to facilitate porting and co-scheduling applications from different domains, rather than implementing a holistic scheduler, MCL provides a scheduling framework in which developers can plug in their own scheduling algorithms.

7 Conclusions & Future Work

In this deliverable we described the initial design of the DAPHNE runtime system and its prototype implementation. Moreover, we discussed the support for distribution primitives and the integration with existing frameworks and libraries.

Our future work will focus on supporting more local and distributed operations, integrating with more hardware devices, besides CPUs and GPUs, implementing more distribution primitives, and integrating the DAPHNE runtime with more frameworks and libraries. In addition, we will work on data locality and communication mechanisms in order to improve the performance further. Finally, the runtime system design will evolve continuously along

with its development during the course of the project and will be updated in the upcoming deliverables D4.2, D4.3 and D4.4.

References

[A+16] M. Abadi et al., "TensorFlow: A System for Large-Scale Machine Learning," in 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), Savannah, GA, Nov. 2016, pp. 265–283.

[B21] M. Boehm. DAPHNE Deliverable D2.1 "Initial System Architecture", 2021.

[B+16] M. Boehm et al., "SystemML: Declarative Machine Learning on Spark," Proc VLDB Endow, vol. 9, no. 13, pp. 1425–1436, Sep. 2016.

[B+21] S. Baunsgaard et al. "ExDRa: Exploratory Data Science on Federated Raw Data". In SIGMOD, 2021. pp 2450-2463.

[BCH92] J. Bruck, R. Cypher, and C. T. Ho. "Multiple message broadcasting with generalized Fibonacci trees". In 4th IEEE Symposium on Parallel and Distributed Processing, 1992.

[CKPT17] D. Cutting, D. Karger, J. Pedersen, and J. Tukey. "Scatter/Gather: a cluster-based approach to browsing large document collections". In Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR '92). pp 318–329.

[D+22] P. Damme et al. "An Open and Extensible System Infrastructure for Integrated Data Analysis Pipelines". In CIDR 2022.

[G+20] R. Gioiosa, B. O. Mutlu, S. Lee, J. S. Vetter, G. Picierro, and M. Cesati, "The Minos Computing Library: Efficient Parallel Programming for Extremely Heterogeneous Systems," in Proceedings of the 13th Annual Workshop on General Purpose Processing Using Graphics Processing Unit, 2020, pp. 1–10.

[IREE21] "IREE: Intermediate Representation Execution Environment". Google, 2021. Accessed: Oct. 15, 2021. [Online]. Available: https://github.com/google/iree

[K02] Vipin Kumar. 2002. "Introduction to Parallel Computing (2nd. ed.)". Addison-Wesley Longman Publishing Co., Inc., USA.

[KBY17] A. Kumar, M. Boehm, and J. Yang. 2017. "Data Management in Machine Learning: Challenges, Techniques, and Systems". In Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17). Pp 1717–1722.

[L+20a] C. Lattner et al., "MLIR: A Compiler Infrastructure for the End of Moore's Law." 2020.

[L+20b] S. Li et al., "PyTorch Distributed: Experiences on Accelerating Data Parallel Training." 2020. In Proceedings of the VLDB Endowment, 2020. Vol. 13, No. 12, pp. 3005-3018.

[L+21] C. Lattner et al., "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2021, pp. 2-14.

[LLS07] J. Langford, L. Li, and A. Strehl, "Vowpal wabbit online learning project," 2007.

[M+21] E. Mutlu et al., "COMET: A Domain-Specific Compilation of High-Performance Computational Chemistry." 2021. CoRR abs/2102.06827 (2021)

[Nod21] "nod.ai – Efficient AI – Low Precision Distributed AI." https://nod.ai/ (accessed Oct. 15, 2021).

[P18] J. Proficz., "Improving all-reduce collective operations for imbalanced process arrival patterns". In J. Supercomput. 74(7): 3071-3092 (2018).

[P+19] A. Paszke et al., "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in Advances in Neural Information Processing Systems (NeurIPS), 2019, pp. 8024–8035.

[Pla21] "PlaidML". PlaidML, 2021. Accessed: Oct. 15, 2021. [Online]. Available: https://github.com/plaidml/plaidml

[PyT21a] "PyTorch, TorchScript and PyTorch JIT | Deep Dive", (2020). Accessed: Oct. 15, 2021. [Online Video]. Available: https://www.youtube.com/watch?v=2awmrMRf0dA

[PyT21b] "PyTorch Mobile." https://pytorch.org/mobile/home/ (accessed Oct. 15, 2021).

[RD14] Juan-Antonio Rico-Gallego, Juan-Carlos Díaz-Martín. "Improving the Performance of the MPI_Allreduce Collective Operation through Rank Renaming". In Proceedings of the First International Workshop on Sustainable Ultrascale Computing Systems (NESUS 2014).

[S94] Y. Saad. "SPARSKIT: a basic tool kit for sparse matrix computations - Version 2", 1994.

[SCR20] Jerome Soumagne, Philip Carns and Robert B. Ross. "Advancing RPC for Data Services at Exascale", in IEEE Data Engineering Bulletin. 43, 23-34 (2020).

[Spa21] "Running Spark: an overview of Spark's runtime architecture - Manning." https://freecontent.manning.com/running-spark-an-overview-of-sparks-runtime-architecture/ (accessed Oct. 15, 2021).

[Ten21] "Distributed training with TensorFlow | TensorFlow Core." https://www.tensorflow.org/guide/distributed_training (accessed Oct. 15, 2021).

[TFRT21] "TFRT: A new TensorFlow runtime — The TensorFlow Blog." https://blog.tensorflow.org/2020/04/tfrt-new-tensorflow-runtime.html (accessed Oct. 15, 2021).

[Tor21] "TorchScript Documentation." https://pytorch.org/docs/stable/jit.html (accessed Oct. 15, 2021).

[Z+16] M. Zaharia et al., "Apache Spark: A Unified Engine for Big Data Processing," Commun. ACM, vol. 59, no. 11, pp. 56–65, 2016.

[ZC13] H. Zhao and J. Canny, "Sparse Allreduce: Efficient Scalable Communication for Power-Law Data". CoRR abs/1312.3020 (2013)